

Block 2, Part 5: Server side: – Using databases with PHP

Peter Thomson (2017), Dave McIntyre (2019) and Stephen Rice (2021, 2022)

1 Introduction

In Part 4 we explored some important aspects of the PHP language used for developing and running a web application. There is still one more major component to understand.

The database is the most efficient way of storing and handling large amounts of data, and sharing the data in a controlled manner. PHP includes tools to work with a wide variety of databases, but here we will focus on the industry standard for web development and web servers; the relational databases MySQL and MariaDB, which use Structured Query Language (SQL).

2 Learning outcomes

On successfully completing this part of Block 2 you should be able to:

- describe the basic tabular structure of the data in relational databases, including the use of unique primary keys
- describe the relationship between a server-side language, such as PHP, and a server-side data tier, such as MySQL or MariaDB
- outline how PHP and HTML may be combined to store data in a database, or retrieve, sort and present data as a simple HTML table, and as part of a web page.
- explain the risks of creating SQL queries containing untrusted data and describe how malicious content may be handled safely.

3 Database basics

A database is a system designed to make the storage and retrieval of data as efficient and as accurate as possible.

There are two main approaches to creating a database:

First, the relational database, where data is highly structured and broken down into tables with rows and columns. The data is stored, manipulated and retrieved (often by linking data from different tables back together again) using Structured Query Language (SQL). SQL is an American National Standards Institute (ANSI) standard. MySQL and MariaDB are two very similar open source relational databases, with common

origins in earlier versions of MySQL. There are many open source and commercial relational database engines available; in this module we'll focus on MariaDB, but most of what we do here would work in the same way with MySQL or other relational databases supporting the SQL standard.

The second approach is to use one of a number of non-relational databases, which store data in ways that are more suited to particular purposes, such as document databases, key/value stores, or graph databases. Collectively, these have become known as NoSQL databases. Document databases can be useful when data is unstructured or relationships are not clearly defined. By avoiding some of the guarantees that relational databases provide, NoSQL databases may be scaled to use multiple servers and multiple locations. This makes them particularly useful for handling large amounts of text, or the data stored by a large social media websites. MongoDB is a widely used open source document database engine.

In this module, we'll look at relational databases.

For Part 5 we are using MariaDB as the database engine. We are running it on the TT284 server with a large number of student accounts. Each student has access to the database using their own password.

In a MariaDB or MySQL database the data is stored in tables. Each table holds data with a common purpose. A table of data can be displayed as a two-dimensional table of rows and columns. A single record in a table will occupy one row.

It is critical to the efficiency of a relational database that an item of data is only stored in one place in the database. For example, a person's name will be recorded only in one row of one table, even though information about them might be stored in several different tables. These other rows of data will all be linked by a column in each table for the unique ID, the primary key (or name) used to identify this person in the database.

The operations for creating a new database and adding and modifying tables are not needed for the day to day running of most websites. Those tools can be abused, however and you need to be careful to prevent a hacker from deleting, changing or copying the database!

Common operations in the day to day running of a database are the following:

- **Create:** Add a new row of data to a table – PHP will send an SQL instruction specifying the table, the columns to fill in, and the values for each column in the new row.
- **Read:** Read zero, one or more rows of data from a table – PHP will send an SQL instruction specifying the table, the columns to read, and criteria that the values in each row must match.
- **Update:** Change values in zero, one or more rows in a table – PHP will send an SQL instruction specifying which row(s) in which table should be updated, and the columns to change in each row. Unique IDs should be used to identify rows wherever possible.
- **Delete:** Delete zero, one or more rows of data from a table – PHP will send an SQL instruction specifying which row(s) in which table should be deleted. Unique IDs should be used to identify rows wherever possible.

This set of operations are often referred to as **CRUD**.

Notice that none of these day to day operations add or remove tables.

4 Tables

A database is simply a container in which we can store data and later retrieve it. Inside a relational database the data is held in a series of 'tables' which, as you might expect, are simple two-dimensional structures consisting of rows and columns. Table 1 shows an example.

Table 1 A simple database table

Neil	Smith	neil.smith@tt381.com
Sue	Jones	s.jones@yellow.co.uk
Michelle	Black	mblack@it.co.uk
Nick	Green	nickg@putput.co.uk

Normally each row of a table will contain a set of values relating to a single data item. In Table 1 each row represents an individual in terms of their first name, surname and email address. A database can contain any number of tables. Each table is given a unique name when it is created so that it can be referred to. A table can have any number of columns and rows (up to a product-dependent limit). Each column is also given a unique name within the table so it, too, can be referred to.

Table 1 represents a set of individuals. So we might call the table 'contacts', the first column 'first_name', the second column 'surname' and the last column 'email'. We can now depict the table, together with these names, as in Table 2.

Table 2 Contacts table

first_name	surname	email
Neil	Smith	neil.smith@tt381.com
Sue	Jones	s.jones@yellow.co.uk
Michelle	Black	mblack@it.co.uk
Nick	Green	nickg@putput.co.uk

In this table, the database software will allow us to perform a range of operations. We can, for example, search the table for a specific person. We can sort the table by, say, surname. We can also add or delete rows and change any value, which we might want to do should someone change their email address.

5 Keys

Operations on a table

It is very common for a database to have many thousands of rows of data in a single table. A column that functions as a key – and there may be more than one – is used to identify and link related records.

Keys

In our example we might define 'surname' to be a key. In large databases, this might not be a good option as there may be many individuals sharing the same surname. If you were to search a telephone book, for example, you would first use the person's surname and then their first initial. Often a key is 'invented' for applications to use: for example, staff working in an organisation can be given a unique identifying staff number. This unique identifier in a database record is known as the 'primary' key.

As another example, Open University students are given a unique student identifier, known as a PI (Personal Identifier). This means that information relating to different individuals can be stored even if they have the same name. We can add a column 'student_number' to the previous example and add some further contacts to demonstrate the use of keys.

Table 3 Contacts table with 'student_number' key

student_number	first_name	Surname	email
1	Neil	Smith	neil.smith@tt381.com
2	Sue	Jones	s.jones@yellow.co.uk
3	Michelle	Black	mblack@it.co.uk
4	Nick	Green	nickg@putput.co.uk
5	Neil	Smith	n.s.smith@ouwebapp.org

A second table that stores the TMA scores for these students might not repeat the student's name, relying only on the student number to identify the student.

Table 4 TMA table with 'student_number' key

student_number	TMA01	TMA02	TMA03
1	90	87	
2	44	55	
3	76		
4	76		
5	97	78	

Since this module is not about database design and matters such as efficiency, we will not spend much more time on the use of keys. While this is an aspect you should be aware of, our focus here is on access and the use of PHP. PHP allows us to connect to a database and perform operations on the database and its tables. We can also create and delete entire databases using PHP.

6 SQL for database operations

To manipulate databases, we send SQL instructions to the database for it to execute on our behalf. SQL is a language all of its own and can be rather complex. Thankfully, we can achieve most of the database operations required for most web applications with just a few SQL commands.


Typical sequence of PHP database operations

MySQL functions allow us to connect to a MariaDB database and then perform actions on tables and rows of data in those tables. There is quite a range of PHP functions available, but these are typically used in a fairly straightforward sequence, thus:

1. Connect to the MariaDB database server.
2. Select a specific database to use.
3. Construct an SQL instruction (also known as a 'query') as a string.
4. Execute the SQL string.
5. If the query is such that data is to be returned, then collect the data (such as for display in a dynamic web page) and return it.
6. Close the connection (this happens automatically when PHP execution is completed).

Some SQL queries will not return data: for example, deleting a table row will only return the number of rows deleted. Other queries, such as searching, will return the data (rows) that are found. Using PHP we can obtain data and present it in a web page. So let's now take a look at how to create and execute such SQL queries in PHP.

Activity 1 Upload the PHP files to the TT284 server

 Allow 15 minutes for this activity

Download the zipped folder of files for Block 2 Part 5 from the resources area.


Unzip the Part 5 files into your Block 2 folder on your own computer. You can edit these files here or upload them to the server and use the editor there, but must upload them to run them.

To run these PHP files, and any that you create, you must upload them all to the TT284 server first. Before uploading these files, you may wish to remove the Part 4 files from the server so it is less cluttered. You can download copies to your local machine first if you wish before deleting them on the server.

Note: the TT284 server only uploads the first 20 files you select, so you should upload in two batches: first the file beginning with 'a_' then all the other files.

Note: don't copy and paste code for these files, from word processed files. They are likely to contain characters that will create errors in your code which are difficult to spot.

Activity 2 Configuring your TT284 database

 Allow 15 minutes for this activity

Note that each of you has a database set up on the TT284 server ready for this part of Block 2, and for TMA 02. Only use this for the purposes outlined here.

Go to the welcome page of your TT284 server at: <https://oucu.tt284.open.ac.uk/>

Remember to replace the string oucu with your own OUCU, in lower-case, otherwise you won't see your TT284 server welcome page.

Look for the line on this welcome page that gives you your database username and password as shown in the Server Guide; these are unique to you. Make a note of them.

The first thing we need to do is modify a PHP file that we will use in subsequent activities.

Use your text editor to open the file **credentials.php** from your Part 5 files.

Edit this file to add your own username and password as displayed by the welcome page of the TT284 server.

```
// TODO: Set these to match your own TT284 database credentials
// Note that it is important to use single quotes for the password variable
$database_user = 'youroucuhere';
$database_password = 'YOURPASSWORDHERE';

// OPTIONAL: Edit these lines if you're using another database
$database_host = 'localhost';
$database_name = $database_user . '_db';
```

Save the file **credentials.php** before then uploading it to the TT284 server. This file provides subsequent activities with the credentials: database server host, database name, database user and database password.

Remember, this file is unique to you, so don't share it with others and don't post it to any forum.

6.1 Connect to the server and select a database

Now we know our credentials, we are ready to carry out the first two steps: to connect to the MariaDB database server and select the database to use.

The code to do this is provided in a file called **connect.php**. In this file we first 'require' our file with our connection details. Then we create a database object, which is a variable that identifies our connection to the database server. To create this database object, we pass our details to the 'mysqli_connect' function in the order host, username and password. Then we call the 'select_db' method of the database object to select our own database on the database server, as shown in the following extract:

```
// Obtain database credentials from credentials.php
require 'credentials.php';
// Connect to server and select database
$database = mysqli_connect($database_host, $database_user, $database_password);
if (!$database) {
    echo '<pre>host: ' . htmlspecialchars($database_host) . '</pre>';
    echo '<pre>user: ' . htmlspecialchars($database_user) . '</pre>';
    echo '<pre>password: ' . htmlspecialchars($database_password) . '</pre>';
    die('Unable to connect to database server!');
}
if (!$database->select_db($database_name)) {
    echo '<pre>name: ' . htmlspecialchars($database_name) . '</pre>';
    die('Unable to select database: ' . $database->error);
}
```

Notice the use of an 'if' construct to check if the result of calling mysqli_connect is 'truthy'. If there is a problem, this function will return false, so this check ensures that execution stops, with a suitable message. The credentials are output for you to check they are as they should be.

To make use of the database object we call 'methods' or functions which are properties of that object. In other languages, methods and properties are accessed using the dot operator, e.g. `inputElement.click()` in JavaScript. In PHP, methods are accessed using the '->' operator, so we call `$database->select_db($database_name)` to call the 'select_db' method passing the name of the database to select. Now, any SQL we ask the database object to execute will use our database.

Activity 3 Checking the database credentials

 Allow 15 minutes for this activity

The **connect.php** file is designed to be required from another PHP file and cannot be executed directly. To test it, we will use a 'Table manager' app.

Upload the file **table-manager.php** to your TT284 server space and also **connect.php**, **helpers.php** and **style.css** alongside the existing **credentials.php**.

Open **table-manager.php** in a browser and you should see a page that starts with:

TT284 Block 2 – Table manager

Show/Hide: Script execution reports

The following tables currently exist:

We will discuss the rest of the page output later. If you click the 'Show/Hide' button, you should see the following extra information about what PHP is doing before it outputs the message about the database being connected:

Executing: table-manager.php

Executing: connect.php and credentials.php

Connected to database: student_db

Processing submitted form data

Reading database tables

Reading table columns (if any)

You must confirm that this file correctly opens your database before you can do any further exercises.

The Table manager app checks to see if the **connect.php** and **credentials.php** exist – if they do not it will output a suitable message, and PHP will also output 'Warning' and a 'Fatal error' messages which appear at the end of the page. If the files do exist, they will be executed and the code we've already seen will check for further errors and try to connect to the database server and select the database.

Connecting to MariaDB requires the `$database_user`, `$database_password`, `$database_host` and `$database_name` variables from **credentials.php** to be set correctly, so the first step in the event of a problem is to check those very carefully.

In particular, the password is complex and it is best to copy and paste it rather than retyping it. Ensure that you assign the password using a single quoted string, and watch out for characters PHP considers 'special' like backslash and dollar.

Once it is working, you can try editing **credentials.php** with incorrect credentials to see how it fails on execution.

In each of the exercises that follow, what you should see if the script has run correctly is indicated. If a script fails, then you need to examine the code for clues about what has gone wrong.

Is there a message displayed that says there has been an error? Look to see what the error is. Is it a missing file? A missing password? Is there a punctuation error in your code? Or a word with the incorrect spelling?

As we've noted before, copying code from a web page or from a PDF or Word document is likely to introduce errors that are hard to spot, as the code can look correct but use the wrong character encoding. Always use the downloaded files as your source.

When looking for issues, do switch the debugging on for that file on the server and do view the source of the HTML that PHP produces in case there are error messages hidden inside a HTML tag; also, when working with JavaScript, open the developer tools in the browser and look in the JavaScript console to see reported errors.

If you seek help on the Block 2 Forum, state if this code is running on the TT284 server. State your own operating system, web browser and the script that you are running, and copy the full error message into your forum post.

6.2 Exploring the Table manager app

Before we look further at the database, we will take note of some key features of the Table manager app that we will reuse in later PHP apps.

Enabling debugging

Two lines of PHP are used to ensure that PHP errors, warnings and notices are output, to help you debug the app. These lines are explained in more detail later in this part.

```
// Allow debugging
error_reporting(E_ALL);
ini_set('display_errors', 1);
```

Defining the SAFE_TO_RUN constant

As we are using the 'require' function described in the previous part to break our application into multiple files, we define this constant so that the files recognised they are being used correctly.

Setting variables to configure the app

Near the start of the code, variables are set to control what the app will do. In the case of the Table manager, its purpose is to create or drop a specific table, so there is one line of code to control which table it creates or drops:

```
// TODO: Change this value to configure the application
$database_table = "tt284_guests"; // name of database table to create/drop
```

Creating the \$url, \$task and other variables used in the app

The app uses HTML forms, and each of these forms needs an 'action' attribute – the URL to submit data to, in this case it should be the URL of the file being executed. This is available in the 'superglobal' \$_SERVER array, but for convenience we copy this into a variable named \$url.

```
// Setup variables used in the application
$url = $_SERVER["PHP_SELF"]; // URL of this page for forms to POST to
$tables = []; // list of names of tables in database
$columns = []; // list of names of columns in database table (if it exists)
$task = ''; // task to carry out in response to form submission
```

The `$task` variable is initially set to an empty string. Later we will read the submitted form data for information about what task to perform, and assign it to this variable, then use it to make decisions about what code to execute:

```
// Read $task from submitted form data
if (!empty($_POST['task'])) {
    $task = $_POST['task'];
}
```

If the HTML form submitted to the page has an input with the name 'task', there will be a corresponding entry in the `$_POST` array, and if that is present, that value of that input will now be assigned to the variable `$task` in our PHP code.

Require the `_x` and `_e` functions from `helpers.php`

As discussed in Part 4, the text data we include in PHP output may be malicious – it is certainly untrusted. The data should not contain HTML tags, so to ensure that any HTML tags it does contain are made safe, we can use `htmlspecialchars()` function to 'escape' key HTML characters. However, this function name is long, so to make our code a little more compact, we require the **helpers.php** file which defines some handy shortcuts, both of which are used in the Table manager code:

```
function _x($variable)
{
    return htmlspecialchars($variable);
}
```

The '`_x`' function escapes a value, allowing us to write `_x($boop)` instead of `htmlspecialchars($boop)` when creating strings in PHP code.

```
function _e($variable, $key = null)
{
    if (is_array($variable) and $key != null) {
        if (!empty($variable[$key])) {
            echo _x($variable[$key]);
        }
    } else {
        echo _x($variable);
    }
}
```

The '`_e`' function echoes a value. It is more complex as it accepts one or two arguments. In the simple case, it allows us to write `<?php _e($boop) ?>` instead of `<?php echo htmlspecialchars($boop) ?>` when outputting a variable inside other HTML. The second argument is used when escaping values inside arrays

and we will use this later.

Show/Hide buttons

This app, and others we will write, present 'Show/Hide' buttons which can be clicked to show and hide parts of the page. This is achieved using a HTML label, a checkbox input element and some advanced CSS. This CSS is described further in **style.css** and uses some advanced CSS selectors which you are not required to understand. What you should understand is how these apps write HTML that works with this CSS. The CSS expects three HTML elements immediately after each other:

1. An input element of type checkbox with the 'class' attribute "collapser".
2. A label element connected to the checkbox by using a 'for' attribute that matches the input element's unique 'id' attribute, containing the text of the 'Show/Hide' button.
3. A HTML element containing the content to show or hide, for example a form or table.

Clicking the label checks and unchecks the checkbox (which is 'visually hidden' using CSS). When the checkbox is not checked, CSS hides the element immediately after the label by applying the rule 'display: none'.

Further, as a special case, any HTML elements with a 'class' attribute containing 'report' appearing after (not just immediately after) are hidden by Show/Hide button with the 'id' attribute "show_reports".

Activity 4 Explore the Table manager code

 Allow 30 minutes for this activity

Edit the file **table-manager.php** on the TT284 server and make the following changes in turn. Save the file and open it in your browser to see the effect of each change. Predict what you think will happen then if your prediction is correct. Undo your change and check the file is working as it was before, then try the next change:

- Comment out the lines to enable debugging and change another line so that it will cause an error, such as changing a 'require' statement to a file that does not exist.
- Change the name of the 'SAFE_TO_RUN' constant where it is defined.
- Change the value of the \$database_table variable and observe where it appears in the page output.
- Add the following PHP code immediately after the helpers file is required: \$unsafe = '<script>alert("unsafe!")</script>'; \$safe = _x(\$unsafe); echo \$safe;
- Add the following PHP code immediately after the helpers file is required: \$unsafe = '<script>alert("unsafe!")</script>'; _e(\$unsafe);
- Add the following PHP code immediately after the helpers file is required: \$unsafe = '<script>alert("unsafe!")</script>'; echo(\$unsafe);
- Change the class attribute of the Show/Hide button to a value other than "collapser" or change the link to **style.css** so that the CSS is no longer included.

You may discuss your findings in the Block 2 Forum.

6.3 Create a table

Now we will use the Table manager app to create a new table. Here is an example of an SQL instruction to create a table:

```
CREATE TABLE tt284_guests (  
    id INT(6) UNSIGNED AUTO_INCREMENT PRIMARY KEY,  
    firstname VARCHAR(30) NOT NULL,  
    lastname VARCHAR(30) NOT NULL,  
    email VARCHAR(50) NOT NULL  
)
```

Observe the following features of this instruction:


- The command is CREATE TABLE.
- The name of the table to create follows the command, then a list of fields between an opening and closing round bracket ().
- Each column definition is separated by a comma, so all but the final column are followed by a comma.
- Columns are defined by specifying a name and type information, which for string values is most often VARCHAR(n) NOT NULL, where 'n' is the maximum length of the string.
- The column named 'id' serves as primary key of the table, and rather than manually setting this value, we ask the database to create a new value for each row by increasing a counter. Each row we add will get a new 'id' value, that does not exist, and has never existed, in the table.

This SQL instruction can be executed using the 'query' method of a database object as follows, assuming the SQL is in the variable \$create_sql:

```
$database->query($create_sql);
```

If this method succeeds, it will return a truthy value: if not, it will return false, so you can check for success using an 'if' construct or by assigning the result to a variable and examining it.

Activity 5(a) Create a table in the database

 Allow 30 minutes to complete both parts (a) and (b) of this activity

Open the Table manager app **table-manager.php** you previously uploaded to the TT284 server.

The Table manager app outputs some SQL like the above. Click the 'create' button in the app to execute this SQL. You should see the newly created table appear in the list of tables that exist in the database, and a list of columns in the table.

If you try to create the table again, you will see an error that the table already exists. You can click the 'drop' button if you want to repeat this process.

Edit the **table-manager.php** file on the server. Look for the code that executes the 'create' SQL and examines the result. Try changing this code to output a different message or try using the ! operator to reverse the effect of the 'if' construct, for example:

```
if (!$database->query($create_sql)) {
    echo '<div class="report always">';
    echo 'NOT Created table "' . _x($database_table) . '"';
    echo '</div>';
} else {
    echo '<div class="report always">';
    echo 'NOT Error creating table: ' . _x($database->error);
    echo '</div>';
}
```

Next, experiment with the CREATE SQL instruction, for example to:

- change the order of the columns
- remove a column
- change the name of a column
- change the length of a column
- add a column of your choice, for example a 'telephone' column of length 12.

After you create the table, use the 'drop' button to delete it again so you can test your new CREATE SQL instruction.

Each time you change the SQL to create the table you need to reload the Table manager app to see its effect. If you click the reload button in your browser it will immediately repeat the 'create' or 'drop' task again. To avoid this, reload the page using the link provided on the page itself.

6.4 Drop a table

If you create a table in error and need to correct it, you cannot readily modify it. The simplest solution is to delete it and then create a new one. Deleting a table is called dropping it. Clearly when a table is dropped, all of the data it contains is irrevocably lost so this must be used with care.

We have already created and dropped a table several times, now we will see how this is done using SQL and PHP. As shown on the page the SQL instruction to drop a table is straightforward, for example:

```
DROP TABLE tt284_guests
```

In PHP this is executed in the same way as the SQL to create a table, by using the 'query' method of a database object as follows, assuming the SQL is in the variable \$drop_sql:

```
$database->query($drop_sql);
```

If this method succeeds it will return a truthy value, if not it will return false, so you can check for success using an 'if' construct or by assigning the result to a variable and examining it.

Activity 5(b) Delete a table in the database

Open the Table manager app **table-manager.php** you previously uploaded to the TT284 server.

The Table manager app outputs some SQL like the above. Click the 'drop' button in the app to execute this SQL. You should see the dropped table no longer appears in the list of tables that exist in the database.

If you try to drop the table again, you will see an error that the table does not exist. You can click the 'create' button if you want to repeat this process.

Now we will examine how the app knows to create or delete the table or to just output the SQL to the page. This is achieved using 'submit' inputs as demonstrated in the following extract:

```
<form method="POST" action="<?php _e($url); ?>">
<input type="submit" name="task" value="create" />
<input type="submit" name="task" value="drop" />
CAREFUL! NO UNDO!
</form>
```

Observe that:

- The '_e' helper function is used to safely echo the URL of the page as the form action (the escaping is necessary because there are ways of getting malicious content into the URL!)
- The submit inputs have the name "task". The value of the submit input is the task to carry out and is assigned to the \$task variable as described earlier.
- There are multiple inputs with the name "task" but only the value that is clicked will be submitted.

The decision to carry out a task has the following pattern, though with more output and error checking:

```
if ($task == 'drop') {
    $database->query($drop_sql);
}
```

If you wish you can try editing the code to change the name of the task from 'drop' to 'delete'. If you click the reload button in your browser it will immediately repeat the previous form submission, as if you clicked the submit button again. To avoid this, reload the page using the link provided on the page itself. You could also test your skills by changing the order of the table columns so that the DROP heading, SQL and submit button are in the first column.

Finally, we will try creating and deleting another table. Locate the line of code near the start of `table-manager.php` that configures the application and changes the database table variable. If you did not already experiment with this variable, try changing it in order to create a new table with a different name, then drop the table to tidy up your database.

6.5 Read table structure

The table manager shows a list of tables in the database and for one specific table, shows a list of columns in that table. We will briefly explore how it does this. The SQL instruction to obtain a list of tables from MariaDB is straightforward:

```
SHOW TABLES
```

The SQL instruction to list the columns in a specific table in MariaDB, for example `'tt284_guests'`, is also straightforward:

```
SHOW COLUMNS FROM tt284_guests
```

As with previous instructions, these are executed using the `'query'` method, but this time we keep the result:

```
$sql = "SHOW TABLES";  
$result = $database->query($sql);
```

This result can be checked to see if it is truthy or not to discover if the SQL was executed. If it is truthy, it is a result object which we can use read rows of data from one by one. We start at the first row (if any) returned by the SQL query and keep reading rows until there are no more left.

This is done by repeatedly calling a method like `'fetch_row'` which returns the next row of data as an array of data indexed by numbers. This means that the first item in an array named `$row` is for example `$row[0]` and the next is `$row[1]` and so on.

When there are no more rows, the `'fetch_row'` method will return false, so we can use a `'while'` construct to iterate through the rows. At the start of each loop iteration, we read the next row, assign it to a variable, and examine that variable to see if it is truthy. Inside the while loop we can make use of the variable, for example to store or output it.


The `SHOW TABLES` instruction returns a single value per row, being the table name as a string. We can append this to an array named `$tables` as follows:

```
$tables = []; // list of names of tables in database  
// Read each row as an array indexed by numbers  
while ($row = $result->fetch_row()) {  
    // Put the first item in each row into the tables array  
    $tables[] = $row[0];  
}
```

We can go through the same process with the result of executing a `SHOW COLUMNS` instruction. Then once we have the values in an array, we can output these using a suitable HTML structure, such as unordered list.

Note: The `SHOW TABLES` and `SHOW COLUMNS` instructions are specific to MariaDB/MySQL. The SQL standard provides ways to obtain the same information, but they are a little more complicated.

Activity 6 Show the columns in a table

 Allow 15 minutes for this activity

Open the Table manager app **table-manager.php** you previously uploaded to the TT284 server and ensure you have used it to create a table.

Observe the list of table names and column names shown on the page, and view the source of the HTML, or inspect it in the developer tools, to see how this list is constructed. Compare this structure with the PHP code:

```
echo '<ul>';
// Loop through each value in $tables
foreach ($tables as $table) {
    echo '<li>' . _x($table) . '</li>';
}
echo '</ul>';
```

This uses the ‘foreach’ construct to iterate through the values in the `$tables` array. For each value (if there are any values), we echo a list item tag, the escaped value of the table name, and a closing list item tag. While it is unlikely the table name will contain a character that has a special meaning in HTML, it's good practice to escape anything that should not be interpreted as HTML. If the `$tables` array is empty, observe that this will output an unordered list with no elements.

The same construct is used to output column names from the `$columns` array. Unlike table names in a database, column names in a table do have a meaningful order, so an ordered list would be more appropriate. Edit the code to output an ordered list (``) rather than unordered list (``) and check the code behaves as you expect.

6.6 Read and present rows of data

Now we have connected to the database and created our table, we will read and present rows of data from the table. The SQL instruction to do this for our example table is:

```
SELECT id, firstname, lastname, email FROM tt284_guests
```

That is:

- SELECT
- A list of column names with a comma between each but none after the last column
- FROM
- The table to read rows from

We have already seen how to execute SQL instructions and read the results for the SHOW TABLES query. The code in this case is the same:

```
$result = $database->query($sql);
```

However, in this case we will read each row as an associative array as follows:

```
while ($row = $result->fetch_assoc()) {
    echo '<p>firstname: ' . $row['firstname'] . '</p>';
    echo '<p>lastname: ' . $row['lastname'] . '</p>';
    echo '<p>email: ' . $row['email'] . '</p>';
}
```

Observe that we use the same 'while' construct as before but this time using 'fetch_assoc' method rather than the 'fetch_row' method. Then we access the values in the row variable using keys that correspond to column names. You may recognise the above example is dangerous because the values are not escaped before output. We will correct this later.

We can also obtain the number of rows in the result using the 'num_rows' property of the result object:

```
echo '<p>row count: ' . $result->num_rows . '</p>';
```

Activity 7 Present rows of data as a table

 Allow 20 minutes for this activity

Upload and run file **table-select.php** on the TT284 server, along with **helpers.php**, **style.css**, **connect.php**, **credentials.php** and **data-table.php**.

When you run this for the first time, you should see output including:

Connected to database: student_db

Executing: data-table.php

Data table

```
$sql == SELECT id, firstname, lastname, email FROM tt284_guests
```



firstname	lastname	email
Suzi	Smith	Suzi.Smith@example.com
1 results		

Each time you reload the page, a new row will be added to the table. If you instead see an error that the table doesn't exist, go back to the Table manager app and ensure that the table exists with the expected name and columns.

Open **table-select.php** and **data-table.php** in your text editor. The first file has many of the features we've seen already in the Table manager app: to set up debugging, set the database table, require helper functions, connect to the database and so on. So that you have some rows to look at, each time you run **table-select.php** it also has code to add a row to the table. Finally, it requires **data-table.php** to query the database and output a HTML table.

We will explore the code to insert a row later, but for now look at the code in **data-table.php** which outputs a HTML table of rows from the database. Look for the code that creates the SQL instruction to execute:

```
// TODO: Change this SQL according to the columns you expect
$sql = "SELECT id, firstname, lastname, email";
$sql = $sql . " FROM $database_table";
```

You can try editing this code to change the order of the columns read from the table (this should make no difference) and to omit a column (you should see the values go missing from the HTML table but no error). Next, we will examine the code to output the table. This runs in 'HTML mode' – it is HTML with fragments of PHP embedded in it, rather than PHP code to echo HTML.

The first row of the table is just three headings:

```
<tr>
  <th>firstname</th>
  <th>lastname</th>
  <th>email</th>
</tr>
```

Then in the body of the table, a 'while' loop is used to output a HTML table row for each database table row:

```
<?php while ($row = $result->fetch_assoc()) { ?>
    <tr>
        <td><?php _e($row, 'firstname') ?></td>
        <td><?php _e($row, 'lastname') ?></td>
        <td><?php _e($row, 'email') ?></td>
    </tr>
<?php } ?>
```

Observe the similarities with the code presented earlier – the use of a ‘while’ loop and of the ‘fetch_assoc’ method, but this time inside PHP fragments. Each column value is output using the ‘_e’ function from helpers.php. This both escapes the output and ensures that there is no error if the array does contain a value for that column. This works by passing two arguments to the ‘_e’ function: the array to output, and the key to the value in the array to output (if present).

As discussed in Part 4, the empty() function allows us to check whether a value exists in an array so that PHP will not output a notice if we try to read a variable that does not exist. We can use this here, along with htmlspecialchars() but that would be a lot of repetitive code, which the helper function saves us having to write.

Compare the HTML output by this script with the PHP in data-table.php. Try editing the file to:

- Change the order of the columns in the HTML table
- Remove a column
- Add a column (you can go back to the Table manager app to do this)

6.7 Insert a row of data

Now we have seen how to select rows of data from a table and present them in a HTML table, we will look at inserting rows into the table. The SQL instruction to do this for our example table is:

```
INSERT INTO tt284_guests (firstname, lastname, email)
VALUES ('Fred', 'Doe', 'fred@example.com')
```

That is:

- INSERT INTO
- The name of the table to insert a row into
- A list of column names, with a comma between each but none after the last column, enclosed in round brackets ()
- VALUES
- A list of values in the same order as the column names, with a comma between each but none after the last value, enclosed in round brackets ()
- Each value in our table is a string, so is quoted using single quotes (unlike JavaScript or PHP, standard SQL does not use double quotes)

We have already seen how to execute SQL instructions and check the result. The code in this case is the same:

```
$result = $database->query($sql);
```

This result can be checked to see if it is truthy or not to discover if the SQL was executed. If it is truthy, we can obtain a count of the rows that were changed, but note this is a property of the database object not of the result:

```
echo $database->affected_rows . ' rows changed';
```

We have already seen code like this in action in the previous activity. If you wish, you can go back to that and edit the SQL in **table-select.php** to insert different values.

Often in PHP we want to insert data into a database table from a form submission. This data is usually in the form of an array, for example:


```
$database_table = "tt284_guests";
$data = [
    'firstname' => 'Suzi',
    'lastname' => 'Smith',
    'email' => 'Suzi.Smith@example.com',
];
```

We can then compose an SQL instruction by carefully concatenating strings and array values together as follows:

```
$sql = "INSERT INTO $database_table (firstname, lastname, email)";
$sql = $sql . " VALUES ('";
$sql = $sql . $data['firstname'];
$sql = $sql . "', '";
$sql = $sql . $data['lastname'];
$sql = $sql . "', '";
$sql = $sql . $data['email'];
$sql = $sql . "')";
```

We can execute this SQL instruction then read the data back out of this database table as we did in the previous example to check whether the data has been saved

Activity 8 Transfer data from a PHP array into a database table

 Allow 20 minutes for this activity

Upload and run file **table-insert.php** on the TT284 server, along with **helpers.php**, **style.css**, **connect.php**, **credentials.php**, **data-table.php** and **insert-row.php**.

When you run this file, you should see output including:

```
Connected to database: student_db
```

```
Executing: insert-row.php
```

```
$sql == INSERT INTO tt284_guests (firstname, lastname, email) VALUES  
( 'Suzi', 'Smith', 'Suzi.Smith@example.com' )
```

```
Executing: data-table.php
```

Data table

```
$sql == SELECT id, firstname, lastname, email FROM tt284_guests
```

Each time you reload the page a new row will be added to the table. If you instead see an error that the table doesn't exist, go back to the Table manager app and ensure that the table exists with the expected name and columns. If you want to clear the table, use the Table manager app to drop and create it again.

Open **table-insert.php** and **insert-row.php** in your text editor. The first file has many of the features we've seen already in the Table manager app, to set up debugging, set the database table, require helper functions, connect to the database and so on. Observe that this file contains code to create an array named `$data` with the data to insert into the table. Then it requires it requires **insert-row.php** to save the data into the database, and just like the previous example, requires **data-table.php** to query the database and output a HTML table.

Try changing the values in the `$data` array to insert a different data into the database table and reload the file to see the effect of your changes.

Now look at the code in **insert-row.php** which creates and executes the SQL instruction to save the data. Observe that this code is much as described earlier. You can try editing the SQL to order the columns and values differently (this should make no difference providing the column name and value match up) or to omit a column (you should find this causes an error as the database will not allow the column to be empty due to the "NOT NULL" in column type).

We have seen how to insert a new row of data into the database table. Note that we did not need to set the value of the id field; the database allocated it for us, due to the `AUTO_INCREMENT` in the column type.

However, we hard coded the rest of the data into our script.

We will go onto read the data from a HTML form submission instead. This introduces risks of malicious content – for example if a value has a single quote in it this will be treated as the end of the value may cause errors or even allow arbitrary SQL instructions to be executed. We will see how to handle this as well.

7 Saving form data to a database

In previous sections we have demonstrated how to:

1. create a HTML form
2. validate form data in a web browser
3. submit data to a web server
4. receive data on a web server
5. add data to a database table
6. read data from a database table
7. present data as a HTML table.

We can demonstrate how to make all these work together.

Creating a single web page solution

It is common to create one PHP file which both contains the form and the PHP to handle its data. When the page first loads it recognises that no data has been sent, so it displays the form; when it is loaded with data present it processes the data instead.

Of the steps above, the first three take place in a web browser – it receives a HTML page containing a form, allows the form to be completed, and submits it to the web server. So our PHP code will start at point 4 and at the end will output a HTML form, effectively returning to step 1.

You are provided with a PHP file named **admin.php** (not **a_admin.php**) containing each of these steps. This has the structure you have already encountered in previous examples. Open the file in your text editor and look for the following additions:

- An `$app_title` variable added in the configuration section, which contains a value to output as the page title and heading.
- A `$task` variable which will be used to decide what to do with data submitted by the web browser.
- A `$data` variable which will hold the data submitted by the web browser.
- An additional required file named **head.php** which outputs the start of the page, including the page title and heading, and a link to a stylesheet.
- An additional required file named **data-form.php**

7.1 Read form data submitted by a web browser

Next, we want to collect any values sent from the form – there won't be any the first time we use this page, but we should still check. These will be in the superglobal `$_POST`. Later we will validate this data. This validated data will then be stored in our own array `$webdata`.

For now, we will assign this data directly to `$data` variable, on the assumption it will be a set of string keys and string values:

```
$data = $_POST;
```

Note that this does not deal with the possibility of receiving an array should the form use checkboxes. We will see ways to handle this possibility later.

7.2 Add submitted data to a database table

This is achieved by requiring and executing the same **insert-row.php** file we already explored, but now it is only required and executed if the `$task` variable has the value 'save':

```
// Task is to save submitted form data to a row in the database table...
if ($task == 'save') {
    // So insert the row
    require "insert-row.php";
}
```

7.3 Read and present data from a database table

This is achieved by requiring and executing the same **data-table.php** file we already explored:

```
// Now we have carried out tasks, output HTML table and form
// Output a table of rows in the database table
require "data-table.php";
```

7.4 Present a HTML data entry form

The HTML code for the form is in the **data-form.php** file, and like the code we already saw in the Task manager app, has a HTML form with the following structure:

```
<form method="POST" action="<?php _e($url); ?>">
    <p>
        <label for="save">Submit:</label>
        <input type="submit" id="save" name="task" value="save" />
        <a class="cancel" href="<?php _e($url) ?>">cancel</a>
    </p>
</form>
```

Observe the following:

- The `$url` variable set in **admin.php** is used to ensure the form submits data to the correct URL.
- A submit input with the name 'task' and the value 'save' is used to present a 'save' button on the form and set the value of `$task` when the form is received.
- For accessibility, a label is associated with the submit button. The 'for' attribute of the label and the 'id' attribute of the submit button must match.

- To allow reloading the page without resubmitting form data, a 'cancel' link is also provided.

The form also has input elements for 'firstname' , 'lastname' and 'email' columns, each with the following pattern:

```
<p>
  <label for="firstname">First name:</label>
  <input
    type="text"
    id="firstname"
    name="firstname"
  />
</p>
```

This code is quite repetitive and it's important to get each part correct. Observe the following

- Each input is of type text.
- Each input has a name that matches the column name – this will be submitted to the server.
- For accessibility, a label is associated with the input. The 'for' attribute of the label and the 'id' attribute of the submit button must match. The 'id' attribute is not submitted to the server.

7.5 Putting it all together

Activity 9 A single page solution to add and list rows

 Allow 30 minutes for this activity

Upload the files used by the app (admin.php, connect.php, credentials.php, data-form.php, data-table.php, head.php, helpers.php, insert-row.php, style.css) to the TT284 server and open **admin.php**. You should see:

TT284 Block 2 - CRUD App (Iteration Zero)

Show/Hide: Script execution reports

```
$task == ''
```

Data table

(as previously)

Data entry form

(a form with three text inputs, a save button and a cancel link)

If you do not see what is expected, use the Show/Hide button to examine script output and pay close attention to any error messages to see what might be causing the problem.

If you see an error that the table doesn't exist, go back to the Table manager app and ensure that the table exists with the expected name and columns. If you want to clear the table, use the Table manager app to drop and create it again.

You can now use the data entry form and save button to add new rows to the database table. Try this out with some values of your choice. Observe the script execution reports, the value of the \$task variable, and the SQL instructions executed as they appear on the page.

Try editing the HTML form inputs in **data-form.php**. You can try changing the labels text, 'id' and 'for' attributes. Observe that these make no difference to the data submitted to the web server. Observe that changing the 'name' attribute means the data is not saved.

Notice that if you reload the page another copy of the data is added. That can be a problem with user impatience and them resending data whilst it is still being processed. In many situations we need to take steps to avoid that, but that is beyond the immediate task.

This app forms the beginning of an app to Create, Read, Update, and Delete data in a database table. So far, we have C and a little of R, and we will go on to add code to fill in the gaps.

8 Building a web application

Writing code with the intention of making as much as possible suitable for reuse can save a lot of work and produce much more reliable code.

In this next example we have set out to make as much of this code as possible reusable, so that we can use it again in the future, by making further use of 'require' from the script that calls the function.

Application structure

To do this we first identified each functionally distinct part and places each of these into its own separate file. These files don't need their own HTML head. They only complete their specified function and then the control of the script returns to the page that required them.

We will create a new iteration of the existing CRUD application example by copying the files to new versions beginning with 'a_' and adding some new files. These files are provided for you in the Block 2 Part 5 resources. Some files we will not change, so these are not renamed: connect.php, credentials.php, helpers.php.

We don't want these files to be able to run on their own, so we add the check for 'SAFE_TO_RUN' at the start of each separate script.

The main file is named **a_admin.php** – open this in your text editor to observe the following additions:

- An `$js_file` variable added in the configuration section, which contains the name of a JavaScript file with form validation code the web browser will execute.
- Variables named `$search`, `$sort` and `$order` which will be used to control the rows shown in the data table.
- An additional required file named **a_read-post.php** which reads data from `$_POST` to `$data` and outputs this to the page.
- An additional required file named **a_search-form.php** which presents and processes data submitted by a search/sort form.

At first sight that seems a lot of files, but by separating the functional tasks, many of them can be reused in multiple pages and so significantly reduce the required work in creating a complete project.

Activity 10 A single page solution with search/sort and client-side validation

 Allow 30 minutes for this activity

Upload all the files use by the app (all files beginning with `a_`, `connect.php`, `credentials.php`, `helpers.php`, `style.css`) to the TT284 server and open **a_admin.php**. You should see:

TT284 Block 2 - CRUD App (Iteration A)

Show/Hide: Script execution reports

Show/Hide: Form data submitted by browser

`$task == ''`

Show/Hide: Search form

Show/Hide: Data from database table

Data table

(as previously)

Show/Hide: Data entry form

As previously, you use the data entry form and save button to add new rows to the database table. You can also now control the order of the data table and search for specific values. Try out each of the features and observe the script execution reports, the submitted form data, the value of the `$task` variable, and the SQL instructions executed as they appear on the page.

One further change is that the data table and data entry (and the new search form) are not always shown. They are hidden using Show/Hide buttons to make the page more manageable – while allowing flexibility for the user to view and interact with this content if they wish.

Observe that each part of the user interface has a submit input which submits a form to control the operation of the app. In the case of the search and save forms this includes the data to search for or add to the database, while the data table has an ‘add’ button that submits only that value. The different sections are shown according to the task flag, set by these submit inputs. As you use the app, observe the relationship between the button text, the value of the task variable, and the content which is shown.

Recall that the Show/Hide buttons work by checking a hidden checkbox. To make a Show/Hide element visible by default, the box is checked by using PHP to add the ‘checked’ attribute to the element, for example:

```
// If there is no task, or if searching, show the table by default
if ($task == '' or $task == 'search') {
    $checked = 'checked';
} else {
    $checked = '';
}
?>
☐
```

8.1 Client-side validation

The app now has client-side validation to improve usability, however this does not improve security – that requires server-side validation which we will consider later.

To enable JavaScript validation, the **a_data-form.php** file has gained some extra code to make use of this, in that the form calls the ‘validateForm’ function when it the ‘submit’ event occurs, i.e. when the user attempts to submit the form:

```
<form method="POST" action="<?php _e($url); ?>" onsubmit="return validateForm()">
```

The ‘return’ keyword ensures that if the ‘validateForm’ function returns a false value, the form submission is cancelled by the web browser.

Each input element gains an ‘onchange’ attribute to call the ‘validate’ function to check its value when it is edited by the user, and an initially empty span element to show validation feedback by setting its ‘innerText’ property:

```
<p>
  <label for="firstname">First name:</label>
  <input
    type="text"
    id="firstname"
```

```

        name="firstname"
        onchange="validate(event.target)"
    />
    <span id="feedback_firstname" class="invalid"></span>
</p>

```

Open the **a_admin.js** file in your text editor, examine the 'validateForm' and 'validate' functions and comments, and try to match the input IDs in the JavaScript code to the HTML of the form in your web browser. Try to enter invalid values for each input and observe the effect as you move through the inputs and attempt to submit the form.

8.2 Save data using a prepared statement

We now make an important security improvement to the application by changing how the SQL INSERT instruction is created. Compare **insert-row.php** and **a_insert-row.php** in your text editor and look for the code that creates the SQL instruction. Observe that much of the SQL is the same:

```

INSERT INTO $database_table (firstname, lastname, email)
VALUES ( ... )

```

But now, rather than add values from the \$data array into the SQL for each column, instead the values are specified as:

```

VALUES (?, ?, ?)

```

This is to eliminate a vulnerability which has been the basis of some significant website attacks, SQL injection.

The essence of the attack is that the perpetrator is asked for input data such as a name, but actually places SQL in that data. When it is combined into a single SQL statement the database cannot differentiate between the legitimate SQL and the injected SQL so executes the malicious content. This can expose other data in the database or allow the attacker to change or delete database content.

To prevent this we use a prepared statement, so we write the SQL and identify it as such leaving the input data to be 'bound' to the statement when it is executed. That way the database knows what is executable SQL and what is data.

The SQL statement is first created with the placeholders for the data values represented by '?' in the SQL statement.

```

$sql = "INSERT INTO $database_table (firstname, lastname, email) VALUES (?, ?, ?)";

```

A statement is then prepared by calling the 'prepare' method of the database object using this SQL

```

$stmt = $database->prepare($sql);

```

This method returns a truthy value if it succeeds, or false if it does not, so the statement can be checked using an 'if' construct and an error reported if it is false.

Then the prepared statement is bound to the values that are being sent to the database using the 'bind_param' method of the statement object:

```
$stmt->bind_param('sss', $data['firstname'], $data['lastname'], $data['email']);
```

The first parameter 'sss' indicates there are three string values to follow, in this case the values of 'firstname', 'lastname' and 'email'. You need to change that to match the number and type of values that follow. For example, if only inserting 'firstname' and 'lastname' then 'ss' would be used to indicate two string values. Instead of 's' for string you would use 'i' if a corresponding value is of type integer.

If the number and type of data types specified here do not match the number and type of placeholders in the prepared statement and the data values presented here, then the data will not be accepted. Again, this method will return false if it failed, so the return value can be checked using an 'if' construct.

This prepared and bound statement is then executed using the 'execute' method of the statement object, which will return false if it failed:

```
$stmt->execute();
```

At this point we have safely executed our SQL instruction using the bound data. Note that it is not possible to retrieve the SQL instruction with the values in it as this never actually exists.

Take some time to locate this code in the file **a_insert-row.php** and check you understand what it does at each stage. You can try to change it by, for example, changing the order of the columns and bound values, or removing one of the columns. Observe what happens if the number of 's' characters in the first parameter to bind the values is incorrect, or the number of values passed.

Finally, for a simple demonstration of why prepared statements are necessary, return to the **admin.php** example from the earlier activity and observe what happens if you try to enter an input with a single quote in it. Compare this with the current example – you will need to disable the client-side validation to test this, by disabling JavaScript in your web browser developer tools, or commenting out the relevant parts of the JavaScript code.

8.3 Search and sort data using SQL

Searching for and sorting data needs a web form interface to enter search terms, and any instruction about the way in which results should be sorted or ordered.

The results of a search can be ordered by any column, or on multiple columns. The order of results can be by ascending or descending value.

This is achieved by adding a WHERE clause to the SQL SELECT query used to select the rows. We can search on a specific value:

```
WHERE lastname = 'Doe'
```

or we can search on part of a string by indicating the rest of the content with the LIKE keyword and the % symbol. The following example would find any row of data where there is a letter 'd' anywhere in the lastname:

```
WHERE lastname LIKE '%d%'
```

LIKE 'd%' would find any record that starts with the letter 'd' and '%d' would find any record that ends with the letter 'd'.

A search can be narrowed down or made more specific by combining conditions using AND and OR operators:

```
WHERE lastname = 'Doe' AND firstname = 'John'
```

```
WHERE lastname = 'Doe' OR firstname = 'John'
```

We can set the order of the output from the SQL SELECT query by adding an ORDER BY clause. We can specify the column that should be used, and the direction in which it should be sorted.

The direction can be either ascending or descending, specified using ASC or DESC.

Putting this all together we get an SQL query as follows (this can be all on one line but is broken up here for clarity):

```
SELECT id, firstname, lastname, email
```

```
FROM tt284_guests
```

```
WHERE lastname LIKE '%d%'
```

```
ORDER BY lastname DESC
```

In our PHP script we need to build up our SQL query by inserting variables into the SQL statement. But as we have seen there is a risk of SQL injection if we insert the values from a web form directly into an SQL instruction.:

To prevent this, just as we did for the SQL to store data, we must use a prepared statement for our search. Prepared statements, however, only protect data in SQL statements, not the SQL keywords or column names. We must insert our own, safe, values for the name of the column to be sorted and the order of sort.

Compare **data-table.php** and **a_data-table.php** in your text editor and look for the code that creates and executes the SQL instruction. Observe that given values for 'search', 'sort' and 'order' based on data submitted by the search form, the following extract will create and execute SQL to select any rows that match the search


value in any of three columns.

```
$sql = "SELECT id, firstname, lastname, email";
$sql = $sql . " FROM $database_table";
$sql = $sql . " WHERE firstname LIKE ? OR lastname LIKE ? OR email LIKE ?";
$sql = $sql . " ORDER BY $sort $order";
$stmt = $database->prepare($sql);
$term = '%' . $search . '%';
$stmt->bind_param('sss', $term, $term, $term);
$stmt->execute();
```

The search value is data and is protected using the prepared statement, but the 'sort' and 'order' values must be inserted directly into the SQL instruction, in the same way as the name of the database table. In the next section we will see how these values are made safe.

8.4 Present a HTML search form

Activity 11 Searching and sorting data from a database

 Allow 30 minutes for this activity

Open the **a_admin.php** example again on the TT284 server and click the Show/Hide button to show the search form. Enter a search term and change the sort criteria and click 'search'. Observe that you are now shown the data table, but if you click the Show/Hide to see the search form again, the input reflects the search (but not the sort) criteria you entered.

This form is output by the **a_search-form.php** file. Open this file in your text editor and examine the code. Look for the following aspects:

Processing form data

If the 'search' task has been requested, the data submitted by the web browser is carefully processed to use it to search the database, and to present it back on the search form. There are two forms in the app, so checking the 'task' flag is necessary to ensure we process the form data for the correct purpose.

The `$search` variable is read from the submitted data. As we saw earlier this will be bound to a prepared statement in **a_data-table.php** to protect against SQL injection:

```
if (!empty($data['search'])) {
    $search = $data['search'];
}
```

The `$sort` variable is set more carefully. Instead of using the form data directly, it is checked for expected values, being the column names, and only set if the value matches a column name, for example:

```
if ($data['sort'] == 'firstname') {  
    $sort = 'firstname';  
}
```

This ensures that the data submitted by the form is never added to an SQL statement. The value 'id' is not checked for, because the 'sort' value is assumed to be 'id' unless otherwise specified (these defaults are set near the start of the main PHP file).

Similarly, the 'order' value is set if the form input matches a specific value:

```
if ($data['order'] == 'DESC') {  
    $order = 'DESC';  
}
```

The value 'ASC' is not checked for, because the 'order' value is assumed to be 'ASC' unless otherwise specified.

Showing the search value on the search form

The search box is a text input. The PHP sets its 'value' attribute value to the value in the 'search' variable. Note that the '_e' helper function is used to protect against JavaScript injection:

```
<p>  
    <label for="search">Search for:</label>  
    <input type="text" name="search" value="<?php _e($search) ?>" />  
</p>
```

We will go on to use this technique to allow us to present data for editing – either because it has been read from the database table or because it has been submitted but failed server-side validation.

The search form also uses a 'select' input and two 'radio' button inputs to control the sort and order values sent to the server. Note that the two radio button inputs have the same name – only the value of the checked button is submitted. The radio button input labels work by enclosing the input in the label rather than matching the label to the input using 'for' and 'id' attributes:

```
<label>  
    <input type="radio" name="order" value="ASC" />  
    Ascending  
</label>
```

This structure is used in this case to simplify presentation of the input elements on the page. You can experiment with this code by changing the order of the input elements in the form, or order of options in the select input. Observe that even if you change the inputs to submit invalid values, those values are

ignored by the code we reviewed earlier. This is important because while we can use HTML and JavaScript to restrict a client such as a web browser to valid data, malicious users are free to bypass these restrictions.

9 Error handling

Error handling in PHP allows the creator of the application to decide what should happen when an error occurs when the program is running. Some of the error reporting systems are designed for use in development but shouldn't be used when a system is in use.

In development it can be useful to see errors and warnings reported in the browser, but in a production system errors should be logged to a file in a secure area, and a suitable message displayed to the user if the application cannot continue. This might be 'Sorry, an error has occurred'. Note that you don't want to provide any details of an error to an attacker. The application should then stop, for example by calling the `die()` function.

To report and display all errors you can insert this code at the start of your main PHP file:

```
<?php
error_reporting(E_ALL);
ini_set('display_errors', 1);
?>
```

To turn off all error reporting:

```
<?php

error_reporting(0);
ini_set('display_errors', 0);

?>
```

To help you out, the 'Enable Debugging' button on the TT284 server will add these lines to the start of any PHP file. It does not matter if they get added more than once.

10 Handling malicious content

Any system connected to the internet can be attacked, and the attack may be directed towards any part of that system. It might be directed at the operating system, at the web browser of users of that system, or at database queries. An attack can lead to a total takeover of the system, or its complete destruction. Or it can syphon off confidential data, or make small but significant changes to data.

An SQL injection attack may add extra code to the data sent from a form or a URL. For example the expected URL might be:

<https://example.com/show-account?username=fred>

But the attacker uses the URL:

`https://example.com/show-account?username='or'test'='test`

If that `$_GET` value is used directly in an SQL query using string concatenations, the SQL might look like:

```
SELECT * FROM usertable WHERE userList = '' or 'test' = 'test'
```

As 'test' will always equal 'test', all the records in the table could be returned. The same applies to data sent as POST.

These superglobal variables should never, ever, be used directly in any code.

Validating or cleaning the data before it gets used is the first stage in preventing such attacks. We can use PHP to verify that data is of the format we expect, that it is not longer than necessary.

The name of the database, table or column or SQL keyword in an SQL query should never directly use data from the user. They should always be set by the script to prevent the injection of a malicious query. For example, the sort order in an SQL query to extract data from a table. The form might submit `sort=asc` but that value must not be used directly in the query to prevent the injection of a malicious instruction. Instead, as shown in **a_search-form.php**, the script will check if the value is as expected before deciding whether set the keyword "ASC" or "DESC" in the SQL query.

Where data from the user must be included in the SQL instruction, prepared statements must be used, where the SQL instruction uses `?` placeholders and value are bound to those placeholders by the database, as demonstrated in **a_insert-row.php**. Prepared statements should be used for all access to the database, not just when saving data, for example when searching for a value as demonstrated in **a_data-table.php**.

Finally, as discussed in Part 4, data from form submissions and the database must be treated as untrusted and 'escaped' when included in HTML output to prevent JavaScript injection. In this part we saw how to create helper functions to achieve this with less PHP code.

11 Summary

In this part we have seen how PHP can be used in combination with a MySQL or MariaDB database to store, retrieve, sort and present data in an HTML web page. This is a very common and very useful skill set. You should be able to think of several applications or websites that use this type of technology and also be able to discuss how they might be used.

Further reading

There is extensive documentation for PHP, SQL and MariaDB available online:

- The PHP Manual is the main, very comprehensive, PHP reference, with many examples, although many of the user comments at the bottom of each page are out of date.

- The PHP database functions we have user are fully documented in the MySQL Improved Extension manual.
- MariaDB provide a complete reference of the SQL statements the server will execute.
- The w3schools SQL Tutorial is also a useful starting point.

Where next?

Block 2 Part 6 will extend our application to allow updating or deleting a row of data. It will validate data again on the server and return it to the user with feedback if errors are found. Finally, you will amend each part of the application to support an additional database table column.
